January 2022

Finding version information for binary files with YARA fingerprinting using a multi-layered approach

Armijn Hemel

Finding version information for binary files with YARA fingerprinting using a multi-layered approach

Abstract

Detecting provenance of binary files can be done by using the YARA pattern matching tool[1]. It is easy to write or generate YARA rules to detect a particular version of a binary file[2], but detection can be time consuming as for some packages there are many versions, meaning there are potentially lots of different rules that need to be applied, with most of them applied while it is already clear that there will never be any successful matches for those rules.

Using multiple scan phases allows doing a coarse check first to determine the overall package using a generic package rule and then zooming in to find the particular version using package/version specific rules.

Keywords

fingerprinting, elf, code provenance, software scanning, open source compliance, yara

Background

When fingerprinting binaries to see what software is used in it most of the time it is necessary to find the exact version of the software used in the binary, while in other cases just having a general idea of what is inside the binary is enough.

When looking at open source software relatively few packages are in widespread use. The Linux distribution that at the time of writing (December 2021) has the most packages is NixOS, which has a bit over 67,000 packages[3]. Other distributions that are popular, such as Fedora and Debian have around 22,000 and 35,000 packages. Of these packages multiple versions are in use, but when it comes to unique packages (so ignoring the version number) the number of packages is relatively low.

For each version multiple packages could have been released during the lifetime of the project. Some packages have only released less than 10 versions (example: bzip2), while others have many more: BusyBox has over 100 releases, and the Linux kernel has had well over 1,000 releases.

Although it is easy to run the YARA rules for every version of every package it unnecessary wastes resources as many packages can already be ruled out in advance, by searching for identifiers (strings, function names, variable names, and so on) that can be found in many or all versions of specific packages only. For example, if it already can be determined in advance that a certain identifier or set of identifiers can only be found in a specific package and not in any other package, then creating rules for those identifiers allows for a coarse filtering. Package and version specific YARA rules can then be applied after this filtering.

Imagine having rules for 50,000 unique packages and a million rules when taking package/version details into account as well. A coarse filter could reduce fingerprinting from scanning with all million

YARA rules it could be reduced to to 50,000 rules, plus the number of version specific rules for the packages that were found. As an example, if for the package that was found in the coarse scan only 100 versions exist, the number of rules applied would be 50,100.

Method

First a significant amount of (or all) versions of a specific package are downloaded and processed, where processing means:

1. extracting the archive for a single version of a package (for example if it is in compressed form, such as a ZIP file)
2. searching all the source code files in the extracted archive
3. extracting function names/variable names/method names/etc. from each source code file using for example the ctags program
4. extracting strings from each source code file using for example the xgettext program

Then when the archives of all versions of a specific package have been processed:

1. find the common subset of strings, function names, variable names, and so on and generate a YARA rule for that. This is the top level YARA rule to recognize the package.
2. generate YARA rules specific to each version of the package. This is the version specific rule.
3. store a mapping between the top level rule and all version specific rules (example: rule “foobar.yara” is the top level rule, and “foobar-1.0.yara”, “foobar-1.1.yara” and “foobar-2.0.yara” are the version specific rules)

If the common subset of strings, function names, variable names and so on is empty when looking at all versions then this can be done for related versions of the package. This could happen if for example the package was significantly refactored and all the function names, variables names, etc were renamed.

After generating these rules detecting the exact version of a binary becomes a two step process:

1. run YARA with the set of rules consisting of all top level rules.
2. for each top level rule for which there matches run the version specific rules
3. report the most promising matches.

Another method could be that instead of the subset of strings, function names, variable names and so on the superset of those identifiers is used to generate the top level YARA rules files.

References

[1] YARA - https://virustotal.github.io/yara/

[2] Using ELF symbols extracted from dynamically linked ELF binaries for fingerprinting - https://www.tdcommons.org/dpubs_series/4441/

[3] https://repology.org/repositories/statistics/total